Uno Game Engine

Qais Jildeh

Table of Contents

Card Class

The Card class is designed using fundamental object-oriented principles. It is declared as an abstract class, allowing specific types of cards to extend it and implement the getCardDetails method. This promotes code reuse and enforces a common structure for all card types. The class encapsulates its fields (color and cardValue) using private access modifiers and provides public getters and setters to control data access, ensuring encapsulation.

Design Patterns Used

The use of an abstract class suggests the Template Method Pattern, where subclasses will define specific behavior while the general structure remains consistent.

Clean Code Principles

- **Meaningful Naming:** The method and variable names are self-explanatory, such as getCardColor and getCardValue, which improve readability.
- **Single Responsibility Principle (SRP):** The class strictly manages card-related attributes and behaviors, avoiding multiple responsibilities.
- **Encapsulation:** Fields are private with controlled access, preventing unintended modifications.
- **DRY (Don't Repeat Yourself):** The class avoids code duplication by using a copy constructor.
- **Minimal Public Interface:** Only necessary methods are exposed, reducing complexity.

Effective Java Defences

- **Item 8: Avoid finalizers and cleaners:** Any type of finalizers or cleaners have not been used by the Card class.
- **Item 10: Obey the General Contract When Overriding equals:** The equals method is properly overridden, ensuring symmetry, transitivity, and consistency. It correctly handles null and type checking using instanceof.
- **Item 11: Always Override hashCode When Overriding equals:** hashCode is overridden using Objects.hash, ensuring consistency with equals.
- **Item 12: Always Override toString:** The toString method provides a concise string representation of the card.

SOLID Principles Defenses

- **Single Responsibility Principle (SRP):** The Card class only manages card attributes and enforces structure through getCardDetails, ensuring a single responsibility.
- **Open/Closed Principle (OCP):** It is open for extension but closed for modification. Subclasses can extend it without modifying its core functionality.
- **Liskov Substitution Principle (LSP):** Any subclass of Card should work as a Card, ensuring correct behavior when replaced in any context.
- **Interface Segregation Principle (ISP):** Since Card is an abstract class, it does not force unnecessary methods on subclasses.
- **Dependency Inversion Principle (DIP):** High-level modules depend on abstractions (Card), not concrete implementations.

Numbered Card, Action Card, Wild Card Classes

The NumberedCard class extends Card, representing a specific type of card with a predefined cardType field. This adheres to the inheritance principle and promotes code reusability by utilizing the super keyword to call the parent constructor.

The ActionCard class extends Card and introduces an additional actionType field. The constructor initializes actionType based on predefined values, allowing action-based behavior while leveraging inheritance.

The WildCard class extends Card, introducing a wildType field. Similar to ActionCard, the constructor assigns predefined values based on cardValue, ensuring proper differentiation between different wild card types.

Clean Code Principles

- **Meaningful Naming:** The method and variable names are self-explanatory, such as getCardColor and getCardValue, which improve readability.
- **Single Responsibility Principle (SRP):** The Card class strictly manages card-related attributes and behaviors, while NumberedCard, ActionCard, and WildCard specify their own types.
- **Encapsulation:** Fields are private with controlled access, preventing unintended modifications.
- **DRY (Don't Repeat Yourself):** The subclasses avoid redundant code by leveraging the parent class's functionality.
- **Minimal Public Interface:** Only necessary methods are exposed, reducing complexity.

Effective Java Defences

- **Item 8: Avoid finalizers and cleaners:** Any type of finalizers or cleaners have not been used by the Card class.
- **Item 10: Obey the General Contract When Overriding equals:** The equals method in Card ensures symmetry, transitivity, and consistency. It correctly handles null and type checking using instanceof.
- **Item 11: Always Override hashCode When Overriding equals:** hashCode is overridden using Objects.hash, ensuring consistency with equals.
- **Item 12: Always Override toString:** The toString method provides a concise string representation of the card.

SOLID Principles Defenses

- **Single Responsibility Principle (SRP):** The Card class only manages card attributes and enforces structure through getCardDetails, while subclasses define specific card types.
- **Open/Closed Principle (OCP):** NumberedCard, ActionCard, and WildCard extend Card without modifying its core functionality, demonstrating openness for extension.
- **Liskov Substitution Principle (LSP):** Any instance of NumberedCard, ActionCard, or WildCard can be used wherever a Card is expected, ensuring correct behavior when replaced in any context.
- **Interface Segregation Principle (ISP):** Since Card is an abstract class, it does not force unnecessary methods on subclasses.

- **Dependency Inversion Principle (DIP):** High-level modules depend on abstractions (Card), not concrete implementations.

DiscardPile Class

Object-Oriented Design in DiscardPile Class

The DiscardPile class follows fundamental object-oriented programming principles:

- **Encapsulation**: The discardPile list is private, and access is controlled through public methods.
- **Abstraction**: The class provides clear methods like addToDiscardPile, getLastThrownCard, and discardPileAsDeck, abstracting implementation details.
- **Singleton Pattern**: It ensures a single instance via a private constructor and synchronized getDiscardPileInstance().
- **Polymorphism**: Implements toString, equals, and hashCode for built-in Java polymorphic behavior.

Design Patterns Used

- **Singleton Pattern**: Ensures only one discard pile instance exists globally.
- **Factory Method (Minor Usage)**: discardPileAsDeck() transforms the discard pile into a new deck.
- **Immutable Pattern (Partially Violated)**: The discardPile list is mutable and exposed via getDiscardPileCards().

Clean Code Principles

- **Meaningful Naming**: Clear method and variable names improve readability.
- **Single Responsibility Principle (SRP)**: Each method has a single responsibility.
- **Encapsulation**: discardPile is private, ensuring controlled data access.
- **DRY (Don't Repeat Yourself)**: Avoids redundant code by reusing functionality.
- **Avoid Returning Null**: getLastThrownCard() could use Optional<Card> instead of null.

Effective Java Defences

- **Item 3 (Enforce Singleton with a Private Constructor or Enum)**: Uses a private constructor but could further secure against reflection-based instantiation.
- **Item 5 (Prefer Dependency Injection)**: Dependency injection would improve testability over the singleton approach.
- **Item 7 (Avoid Finalizers and Cleaners)**: Properly manages memory without finalizers.
- **Item 10 & 11 (Override equals and hashCode Consistently)**: Ensures proper object comparisons.
- **Item 12: Always Override toString**: The toString method provides a concise string representation of the discard pile.

SOLID Principles Defenses

- **Single Responsibility Principle (SRP)**: Manages discard pile operations without additional responsibilities.

- **Open/Closed Principle (OCP)**: Hard to extend; modifications require changes in core logic. However, a discard pile always operates in a fixed manner.
- **Liskov Substitution Principle (LSP)**: Not directly applicable but maintains expected behavior.
- **Interface Segregation Principle (ISP)**: Could implement an interface for better abstraction.
- **Dependency Inversion Principle (DIP)**: Directly manages ArrayList<Card> instead of using an abstraction.

Deck Class

Object-Oriented Design in Deck Class

The Deck class is designed using fundamental OOP principles:

- **Encapsulation**: The unoDeck list is private, ensuring controlled access via methods.
- **Abstraction**: Methods like initializeDeck, shuffleDeck, and drawFromDeck abstract implementation details.
- **Singleton Pattern**: Ensures a single instance through a private constructor and synchronized getDeckInstance().
- **Polymorphism**: Implements toString, equals, and hashCode to interact seamlessly with Java collections.

Design Patterns Used

- **Singleton Pattern**: Ensures only one deck instance exists globally.
- **Factory Method (Implicitly Used)**: initializeDeck() acts as a factory, generating and assembling different card types.
- **Strategy Pattern (Potential Improvement)**: Could be used to handle different shuffling algorithms.

Clean Code Principles

- **Meaningful Naming**: Method and variable names clearly describe their purpose.
- **Single Responsibility Principle (SRP)**: Deck handles card storage, initialization, and shuffling.
- **Encapsulation**: The deck list is private, but getDeckCards() exposes it directly (should return an unmodifiable list).
- **DRY (Don't Repeat Yourself)**: Numbered, action, and wild card initialization methods prevent redundancy.
- **Avoid Returning Null**: drawFromDeck() should return Optional<Card> instead of null.

Effective Java Defences

- **Item 3 (Enforce Singleton with a Private Constructor or Enum)**: Uses a private constructor but could further secure against reflection-based instantiation.
- **Item 5 (Prefer Dependency Injection)**: Using dependency injection instead of a singleton improves testability.

- **Item 7 (Avoid Finalizers and Cleaners)**: Properly manages memory without using finalizers.
- **Item 10 & 11 (Override equals and hashCode Consistently)**: Implements equals and hashCode to ensure proper object comparison.
- **Item 12: Always Override toString**: The toString method provides a concise string representation of the deck.

SOLID Principles Defenses

- **Single Responsibility Principle (SRP)**: Deck creation, storage, and manipulation are well-structured into separate methods.
- **Open/Closed Principle (OCP)**: Extending functionality (e.g., new shuffling algorithms) requires modifying the class instead of extending it.
- **Liskov Substitution Principle (LSP)**: Not directly applicable but maintains expected behavior.
- **Interface Segregation Principle (ISP)**: Could implement an interface for better abstraction.
- **Dependency Inversion Principle (DIP)**: Directly depends on concrete ArrayList<Card> instead of using an abstraction.

Player Class

Object-Oriented Design in Player Class

The Player class follows essential OOP principles:

- **Encapsulation**: The player's hand is stored privately and accessed through controlled methods.
- **Abstraction**: Methods like drawCard, playCard, and printPlayerHand encapsulate complex operations.
- **Polymorphism**: Implements toString, equals, and hashCode for proper object interactions.

Clean Code Principles

- **Meaningful Naming**: Method and variable names clearly describe their purpose.
- **Single Responsibility Principle (SRP)**: The class manages player details and card interactions separately.
- **Encapsulation**: hand is private, preventing direct modification.
- **Cohesion**: The class is well-structured, keeping player-related logic together.

Effective Java Defences

- **Item 7 (Avoid Finalizers and Cleaners)**: Properly manages memory without using finalizers.
- **Item 10 & 11 (Override equals and hashCode Consistently)**: Implements these methods properly to ensure correct behavior in collections.
- **Item 12 (Always Override toString)**: Provides a readable representation of the player and their hand.

SOLID Principles Defenses

- **Single Responsibility Principle (SRP)**: The class focuses solely on managing a player's cards and name.
- **Open/Closed Principle (OCP)**: Can extend behavior without modifying existing code.
- **Liskov Substitution Principle (LSP)**: Any subclass of Player would function correctly if introduced.
- **Interface Segregation Principle (ISP)**: Could implement an interface for better abstraction.
- **Dependency Inversion Principle (DIP)**: The class depends on Card, a concrete class; using an interface like ICard could improve flexibility.

Game Class

Object-Oriented Design in the Game Class

The Game class follows fundamental OOP principles:

- **Encapsulation**: protected fields like deck, discardPile, and game state flags (e.g., skipFlag, reverseFlag, drawTwoFlag) ensure controlled access through methods outside the class while maintaining access for any subclasses extending the Game class.
- **Abstraction**: Methods such as initializeGame, distributeCards, play, and updateNextPlayerIndex encapsulate complex game operations.
- **Polymorphism**: Implements toString and equals to ensure proper interactions between game objects.
- **Singleton Usage**: Deck and DiscardPile use the Singleton pattern, ensuring only one instance of each exists.

Design Patterns Used

- **Singleton Pattern**: Ensures a single instance of Deck and DiscardPile through getDeckInstance() and getDiscardPileInstance().

Clean Code Principles

- **Meaningful Naming**: Methods like initializeGame, distributeCards, and updateNextPlayerIndex clearly describe their purpose.
- **Single Responsibility Principle (SRP)**: Game handles overall gameplay, while Player and Deck manage their respective roles.
- **Encapsulation**: Flags like skipFlag and drawFourFlag are private, preventing external modifications.
- **Cohesion**: All methods contribute to the game flow without unnecessary dependencies.

Effective Java Defences

- **Item 7 (Avoid Finalizers and Cleaners)**: Properly manages memory without using finalizers.
- **Item 10 & 11 (Override equals and hashCode Consistently)**: Ensures proper object comparisons.
- **Item 12 (Always Override toString)**: Provides a readable representation of the game state.

SOLID Principles Defences

- **Single Responsibility Principle (SRP)**: Game manages gameplay, while Deck and Player handle their respective tasks.
- **Open/Closed Principle (OCP)**: The game logic is structured to allow new rules or mechanics without modifying existing code.
- **Liskov Substitution Principle (LSP)**: Any subclass of Game should function correctly.
- **Dependency Inversion Principle (DIP)**: Game depends on concrete classes (Deck, DiscardPile).

GameDriver Class

The game driver class contains the main method and only two lines of code: the first to instantiate a game object (which extends the Game Class) and the second to invoke the play method.